```
DIRECTORY
    ImageDefs: FROM "ImageDefs" USING [MakeImage, StopMesa],
    IODefs: FROM "IODefs" USING [GetInputStream, ReadLine, SetInputStream, TAB, WriteChar, WriteString, WriteLine],
    SegmentDefs: FROM "SegmentDefs" USING [FileNameError],
    StreamDefs: FROM "StreamDefs" USING [NewByteStream, Read, StreamError, StreamHandle],
    StringDefs: FROM "StringDefs" USING [AppendChar, AppendString, EquivalentString],
    SystemDefs: FROM "SystemDefs" USING [AllocateHeapNode, AllocateHeapString];

Red: PROGRAM IMPORTS ImageDefs, IODefs, SegmentDefs, StreamDefs, StringDefs, SystemDefs =
BEGIN

EKind: TYPE = {atom, seq, ap};

ERep: TYPE = RECORD [
    SELECT eKind: EKind FROM
        atom => [
            link: Atom,
            pname: STRING,
            binding: SELECT bKind: * FROM
                unbound => NULL,
                primRegular => [primitive: Regular],
                primMeta => [primitive: Meta],
                defined => [definition: E],
                ENDCASE],
        seq => [
            SELECT sKind: * FROM
                nil => NULL,
                nonNil => [first: E, rest: Seq],
                ENDCASE],
        ap => [rator: E, rand: E],
        ENDCASE];

Atom: TYPE = POINTER TO atom ERep;
Nil: TYPE = POINTER TO nil seq ERep;
NNSeq: TYPE = POINTER TO nonNil seq ERep;
Seq: TYPE = POINTER TO seq ERep;
Ap: TYPE = POINTER TO ap ERep;
E: TYPE = POINTER TO ERep;

Regular: TYPE = PROCEDURE [operand: E] RETURNS [E];
Meta: TYPE = PROCEDURE [parameters: Seq, operand: E] RETURNS [E];

atomList: Atom ← LOOPHOLE[NIL];

true: Atom = MkAtom["true"];
false: Atom = MkAtom["false"];

nilRep: nil seq ERep ← ERep[seq[nil[]]];

nil: Nil = @nilRep;

MkAtom: PROCEDURE [pname: STRING] RETURNS [a: Atom] =
    BEGIN
    FOR a ← atomList, a.link WHILE a~ =NIL DO
        IF StringDefs.EquivalentString[pname, a.pname] THEN EXIT;
        REPEAT FINISHED =>
            BEGIN
            a ← LOOPHOLE[SystemDefs.AllocateHeapNode[SIZE[atom ERep]]];
            a↑ ← ERep[atom[atomList, SystemDefs.AllocateHeapString[pname.length], unbound[]]];
            StringDefs.AppendString[a.pname, pname];
            atomList ← a
            END
        ENDLOOP
    END;

MkSeq: PROCEDURE [first: E, rest: Seq] RETURNS [s: NNSeq] =
    BEGIN
    s ← LOOPHOLE[SystemDefs.AllocateHeapNode[SIZE[nonNil seq ERep]], NNSeq];
    s↑ ← ERep[seq[nonNil[first, rest]]]
    END;

MkAp: PROCEDURE [rator: E, rand: E] RETURNS [a: Ap] =
    BEGIN
    a ← LOOPHOLE[SystemDefs.AllocateHeapNode[SIZE[ap ERep]], Ap];
    a↑ ← ERep[ap[rator, rand]]
    END;
```

```
SeqFromPair: PROCEDURE [e1, e2: E] RETURNS [NNSeq] = BEGIN RETURN[MkSeq[e1, MkSeq[e2, nil]]] END;

TVFromBool: PROCEDURE [b: BOOLEAN] RETURNS [Atom] = BEGIN RETURN[IF b THEN true ELSE false] END;

BoolFromE: PROCEDURE [expression: E] RETURNS [BOOLEAN] =
    BEGIN
    RETURN[SELECT AtomFromE[expression ! Error => ERROR Error["non truth value"]] FROM
        true => TRUE,
        false => FALSE,
        ENDCASE => ERROR Error["non truth value"]]
    END;

UnitFromSeq: PROCEDURE [sequence: Seq] RETURNS [E] =
    BEGIN
    nns: NNSeq = NNSeqFromSeq[sequence];
    IF nns.rest = nil THEN RETURN[nns.first] ELSE ERROR Error["too long"]
    END;

PairFromSeq: PROCEDURE [sequence: Seq] RETURNS [E, E] =
    BEGIN
    nns: NNSeq = NNSeqFromSeq[sequence];
    RETURN[nns.first, UnitFromSeq[nns.rest]]
    END;

TripleFromSeq: PROCEDURE [sequence: Seq] RETURNS [E, E, E] =
    BEGIN
    nns: NNSeq = NNSeqFromSeq[sequence];
    e2, e3: E; [e2, e3] ← PairFromSeq[nns.rest];
    RETURN[nns.first, e2, e3]
    END;

AtomFromE: PROCEDURE [expression: E] RETURNS [Atom] =
    BEGIN
    RETURN[WITH e: expression SELECT FROM
        atom => @e,
        ENDCASE => ERROR Error["non atom"]]
    END;

SeqFromE: PROCEDURE [expression: E] RETURNS [Seq] =
    BEGIN
    RETURN[WITH e: expression SELECT FROM
        seq => @e,
        ENDCASE => ERROR Error["non seq"]]
    END;

NNSeqFromSeq: PROCEDURE [sequence: Seq] RETURNS [NNSeq] =
    BEGIN
    RETURN[WITH s: sequence SELECT FROM
        nil => ERROR Error["too short"],
        nonNil => @s,
        ENDCASE => ERROR]
    END;

NNSeqFromE: PROCEDURE [expression: E] RETURNS [NNSeq] =
    BEGIN
    RETURN[NNSeqFromSeq[SeqFromE[expression]]]
    END;

PairFromE: PROCEDURE [expression: E] RETURNS [e1, e2: E] =
    BEGIN
    [e1, e2] ← PairFromSeq[SeqFromE[expression]];
    RETURN[e1, e2]
    END;

MapSeq: PROCEDURE [sequence: Seq, f: PROCEDURE [E] RETURNS [E]] RETURNS [result: Seq] =
    BEGIN
    old, new: NNSeq;
    WITH s: sequence SELECT FROM
        nil => result ← sequence;
        nonNil =>
            BEGIN ENABLE Marking => Register[result];
            old ← @s;
            new ← result ← MkSeq[f[old.first], nil];
            DO
                WITH r: old.rest SELECT FROM
```

```
                    nil => EXIT;
                    nonNil => old ← @r;
                    ENDCASE;
                new ← new.rest ← MkSeq[f[old.first], nil]
                ENDLOOP
            END;
        ENDCASE
    END;

Eval: PROCEDURE [expression: E] RETURNS [E] =
    BEGIN
    RETURN[WITH e: expression SELECT FROM
        atom => expression,
        seq => MapSeq[@e, Eval ! Unwind => ERROR Unwind[TRUE]],
        ap => Apply[Eval[e.rator], Eval[e.rand]],
        ENDCASE => ERROR]
    END;

Apply: PROCEDURE [operator: E, operand: E] RETURNS [E] =
    BEGIN
    metaRator, metaRand: E;
    BEGIN ENABLE
        BEGIN
        Error => BEGIN PrintError[descriptor, MkAp[operator, operand]]; GO TO StartUnwind END;
        Unwind => BEGIN PrintContext[subexpression, MkAp[operator, operand]]; ERROR Unwind[FALSE] END
        END;
    WITH rator: operator SELECT FROM
        atom =>
            WITH atomRator: rator SELECT FROM
                unbound => ERROR Error["unbound atom"];
                primRegular => RETURN[atomRator.primitive[operand]];
                primMeta =>
                    BEGIN
                    [metaRator, metaRand] ← PairFromE[operand];
                    RETURN[atomRator.primitive[NNSeqFromE[metaRator], metaRand]]
                    END;
                defined => RETURN[Apply[atomRator.definition, operand]];
                ENDCASE;
        seq =>
            WITH seqRator: rator SELECT FROM
                nil => ERROR Error["nil seq operator"];
                nonNil =>
                    WITH srf: seqRator.first SELECT FROM
                        atom =>
                            WITH srf SELECT FROM
                                unbound => ERROR Error["unbound atom"];
                                primRegular => RETURN[primitive[SeqFromPair[@seqRator, operand]]];
                                primMeta => RETURN[primitive[seqRator.rest, operand]];
                                defined => RETURN[Apply[MkSeq[definition, seqRator.rest], operand]];
                                ENDCASE;
                        seq =>
                            RETURN[Apply[@srf, SeqFromPair[@seqRator, operand]]];
                        ENDCASE => ERROR;
                ENDCASE;
        ENDCASE => ERROR
    EXITS StartUnwind => ERROR Unwind[FALSE]
    END;
    RETURN[ERROR] -- to make the compiler happy
    END;

Error: ERROR [descriptor: STRING] = CODE;

Unwind: ERROR [subexpression: BOOLEAN] = CODE;

And: Regular =
    BEGIN e1, e2: E; [e1, e2] ← PairFromE[operand];
    RETURN[TVFromBool[BoolFromE[e1] AND BoolFromE[e2]]]
    END;

RApply: Regular =
    BEGIN rator, rand: E; [rator, rand] ← PairFromE[operand];
    RETURN[Apply[rator, rand]]
    END;
```

```
RAtom: Regular =
    BEGIN
    RETURN[TVFromBool[operand.eKind = atom]]
    END;

Define: Regular =
    BEGIN
    name, value: E;
    n: Atom;
    [name, value] ← PairFromE[operand];
    n ← AtomFromE[name ! Error => ERROR Error["non atom first"]];
    WITH v: value SELECT FROM
        atom =>
            WITH v SELECT FROM -- in lieu of entire variant as RightSide:
                unbound => n.binding ← unbound[];
                primRegular => n.binding ← primRegular[primitive];
                primMeta => n.binding ← primMeta[primitive];
                defined => n.binding ← defined[definition];
                ENDCASE;
        seq => n.binding ← defined[@v];
        ENDCASE => ERROR;
    RETURN[operand]
    END;

Distl: Regular =
    BEGIN first, second: E;
    PairL: PROCEDURE [e: E] RETURNS [E] = BEGIN RETURN[SeqFromPair[first, e]] END;
    [first, second] ← PairFromE[operand];
    RETURN[MapSeq[SeqFromE[second], PairL]]
    END;

Distr: Regular =
    BEGIN first, second: E;
    PairR: PROCEDURE [e: E] RETURNS [E] = BEGIN RETURN[SeqFromPair[e, second]] END;
    [first, second] ← PairFromE[operand];
    RETURN[MapSeq[SeqFromE[first], PairR]]
    END;

Eq: Regular =
    BEGIN
    EqPair: PROCEDURE [expression1, expression2: E] RETURNS [BOOLEAN] =
        BEGIN
        seq1, seq2: Seq;
        WITH e1: expression1 SELECT FROM
            atom =>
                WITH e2: expression2 SELECT FROM
                    atom => RETURN[@e1 = @e2];
                    seq => RETURN[FALSE];
                    ENDCASE => ERROR;
            seq =>
                WITH e2: expression2 SELECT FROM
                    atom => RETURN[FALSE];
                    seq =>
                        BEGIN
                        seq1 ← @e1; seq2 ← @e2;
                        DO
                            WITH s1: seq1 SELECT FROM
                                nil =>
                                    WITH s2: seq2 SELECT FROM
                                        nil => RETURN[TRUE];
                                        nonNil => RETURN[FALSE];
                                        ENDCASE;
                                nonNil =>
                                    WITH s2: seq2 SELECT FROM
                                        nil => RETURN[FALSE];
                                        nonNil =>
                                            BEGIN
                                            IF ~EqPair[s1.first, s2.first] THEN RETURN[FALSE];
                                            seq1 ← s1.rest; seq2 ← s2.rest
                                            END;
                                        ENDCASE;
                                ENDCASE
                            ENDLOOP
```

```
                        END;
                    ENDCASE => ERROR;
                ENDCASE => ERROR
            END;
        e1, e2: E; [e1, e2] ← PairFromE[operand];
        RETURN[TVFromBool[EqPair[e1, e2]]]
        END;

    I: Regular =
        BEGIN
        RETURN[operand]
        END;

    JoinI: Regular =
        BEGIN
        first, second: E; [first, second] ← PairFromE[operand];
        RETURN[MkSeq[first, SeqFromE[second ! Error => ERROR Error["non seq second"]]]]
        END;

    Not: Regular =
        BEGIN
        RETURN[TVFromBool[NOT BoolFromE[operand]]]
        END;

    Null: Regular =
        BEGIN
        RETURN[TVFromBool[operand = nil]]
        END;

    One: Regular =
        BEGIN
        RETURN[NNSeqFromE[operand].first]
        END;

    Or: Regular =
        BEGIN e1, e2: E; [e1, e2] ← PairFromE[operand];
        RETURN[TVFromBool[BoolFromE[e1] OR BoolFromE[e2]]]
        END;

    Print: Regular =
        BEGIN
        PrintE[operand]; IODefs.WriteLine[""];
        RETURN[operand]
        END;

    Quit: Regular =
        BEGIN
        ImageDefs.StopMesa[];
        RETURN[ERROR] -- to make the compiler happy
        END;

    Read: Regular =
        BEGIN
        a: Atom = AtomFromE[operand];
        strmOld: StreamDefs.StreamHandle = IODefs.GetInputStream[];
        strmNew: StreamDefs.StreamHandle = StreamDefs.NewByteStream[name: a.pname, access: StreamDefs.Read !
                SegmentDefs.FileNameError => ERROR Error["nonexistent file to read"]];
        CleanUp: PROCEDURE = BEGIN IODefs.SetInputStream[strmOld]; strmNew.destroy[strmNew] END;
        e: E;
            BEGIN ENABLE UNWIND => CleanUp[];
            IODefs.SetInputStream[strmNew];
            e ← ReadE[ !
                    SyntaxError => ERROR Error["syntax error in file to read"];
                    StreamDefs.StreamError => IF error = StreamEnd THEN ERROR Error["syntax error in file to read"]];
            CleanUp[]
            END;
        RETURN[Eval[e]]
        END;

    Save: Regular =
        BEGIN
        ImageDefs.MakeImage[AtomFromE[operand].pname];
        RETURN[operand]
        END;
```

```
Sepl: Regular =
    BEGIN nns: NNSeq = NNSeqFromE[operand];
    RETURN[SeqFromPair[nns.first, nns.rest]]
    END;

Tail: Regular =
    BEGIN
    RETURN[NNSeqFromE[operand].rest]
    END;

Aa: Meta =
    BEGIN
    f: E = UnitFromSeq[parameters ! Error => Error["wrong number of parameters"]];
    ApplyToElt: PROCEDURE [e: E] RETURNS [E] = BEGIN RETURN[Apply[f, e ! Unwind => ERROR Unwind[TRUE]]] END;
    RETURN[MapSeq[SeqFromE[operand], ApplyToElt]]
    END;

C: Meta =
    BEGIN
    RETURN[UnitFromSeq[parameters ! Error => Error["wrong number of parameters"]]]
    END;

Cn: Meta =
    BEGIN
    p, f, g: E; [p, f, g] ← TripleFromSeq[parameters ! Error => Error["wrong number of parameters"]];
    RETURN[Apply[IF BoolFromE[Apply[p, operand ! Unwind => ERROR Unwind[TRUE]]] THEN f ELSE g, operand]]
    END;

Comp: Meta =
    BEGIN
    RETURN[WITH parameters SELECT FROM
        nil => operand,
        nonNil => Apply[first, Comp[rest, operand] ! Unwind => ERROR Unwind[TRUE]],
        ENDCASE => ERROR]
    END;

Insert: Meta =
    BEGIN
    f: E = UnitFromSeq[parameters];
    nns: NNSeq ← NNSeqFromE[operand];
    result: E ← nns.first;
    DO ENABLE Unwind => ERROR Unwind[TRUE];
        WITH tail: nns.rest SELECT FROM
            nil => RETURN[result];
            nonNil => BEGIN result ← Apply[f, SeqFromPair[result, tail.first]]; nns ← @tail END;
            ENDCASE
        ENDLOOP
    END;

Nf: Meta =
    BEGIN
    ApplyEltTo: PROCEDURE [f: E] RETURNS [E] =
        BEGIN RETURN[Apply[f, operand ! Unwind => ERROR Unwind[TRUE]]] END;
    RETURN[MapSeq[parameters, ApplyEltTo]]
    END;

While: Meta =
    BEGIN
    result: E ← operand;
    p, f: E; [p, f] ← PairFromSeq[parameters ! Error => ERROR Error["wrong number of parameters"]];
    BEGIN ENABLE Unwind => ERROR Unwind[TRUE];
    WHILE BoolFromE[Apply[p, result]] DO result ← Apply[f, result] ENDLOOP;
    RETURN[result]
    END
    END;

TKind: TYPE = {atom, beginSeq, delimSeq, endSeq, beginAp, delimAp, endAp};
Token: TYPE = RECORD [SELECT kind: TKind FROM
    atom => [body: Atom],
    beginSeq, delimSeq, endSeq,
    beginAp, delimAp, endAp => NULL,
    ENDCASE];

ReadTokenLine: STRING = [200];
ReadTokenLineEmpty: BOOLEAN ← TRUE;
```

```
ReadTokenLineIndex: CARDINAL;
ReadTokenString: STRING = [40];
ReadTokenToken: Token;
ReadTokenTokenPresent: BOOLEAN ← FALSE;

ReadE: PROCEDURE RETURNS [expression: E] =
    BEGIN
    s: NNSeq;
    f, a: E;
    ReadToken: PROCEDURE RETURNS [t: Token] =
        BEGIN
        ch: CHARACTER;
        ReadChar: PROCEDURE RETURNS [ch: CHARACTER] =
            BEGIN
            WHILE ReadTokenLineEmpty OR ReadTokenLineIndex = ReadTokenLine.length DO
                IODefs.ReadLine[ReadTokenLine];
                ReadTokenLineIndex ← 0;
                ReadTokenLineEmpty ← FALSE
                ENDLOOP;
            ch ← ReadTokenLine[ReadTokenLineIndex];
            ReadTokenLineIndex ← ReadTokenLineIndex + 1;
            RETURN[ch]
            END;
        UnreadChar: PROCEDURE =
            BEGIN ReadTokenLineIndex ← ReadTokenLineIndex-1; RETURN END;
        -- Body of ReadToken
        IF ReadTokenTokenPresent THEN BEGIN ReadTokenTokenPresent ← FALSE; RETURN[ReadTokenToken] END;
        DO
            SELECT ch ← ReadChar[] FROM
                ' , IODefs.TAB => NULL;
                '< => BEGIN t ← Token[beginSeq[]]; EXIT END;
                ', => BEGIN t ← Token[delimSeq[]]; EXIT END;
                '> => BEGIN t ← Token[endSeq[]]; EXIT END;
                '( => BEGIN t ← Token[beginAp[]]; EXIT END;
                ': => BEGIN t ← Token[delimAp[]]; EXIT END;
                ') => BEGIN t ← Token[endAp[]]; EXIT END;
                ENDCASE =>
                    BEGIN
                    ReadTokenString.length ← 1; ReadTokenString[0] ← ch;
                    DO
                        SELECT ch ← ReadChar[] FROM
                            ' , IODefs.TAB, '<, ',, '>, '(, ':, ') => EXIT;
                            ENDCASE => StringDefs.AppendChar[ReadTokenString, ch]
                        ENDLOOP;
                    UnreadChar;
                    t ← Token[atom[MkAtom[ReadTokenString]]];
                    EXIT
                    END
            ENDLOOP;
        ReadTokenToken ← t;
        RETURN[t]
        END;

    UnreadToken: PROCEDURE =
        BEGIN IF ReadTokenTokenPresent THEN ERROR; ReadTokenTokenPresent ← TRUE; RETURN END;
    t: Token ← ReadToken[];
    -- Body of ReadE
    WITH t SELECT FROM
        atom => expression ← body;
        beginSeq =>
            BEGIN
            IF ReadToken[].kind = endSeq THEN RETURN[nil] ELSE UnreadToken[];
            expression ← s ← MkSeq[ReadE[], nil];
            DO
                SELECT ReadToken[].kind FROM
                    delimSeq => s ← s.rest ← MkSeq[ReadE[], nil];
                    endSeq => EXIT;
                    ENDCASE => ERROR SyntaxError
                ENDLOOP
            END;
        beginAp =>
            BEGIN
```

```
            f ← ReadE[];
            IF ReadToken[].kind~ = delimAp THEN ERROR SyntaxError;
            a ← ReadE[];
            IF ReadToken[].kind~ = endAp THEN ERROR SyntaxError;
            expression ← MkAp[f, a]
            END;
        ENDCASE => ERROR SyntaxError;
    RETURN[expression]
    END;

SyntaxError: ERROR = CODE;

PrintE: PROCEDURE [expression: E] =
    BEGIN OPEN IODefs;
    firstTime: BOOLEAN ← TRUE;
    WITH e: expression SELECT FROM
        atom => WriteString[e.pname];
        seq =>
            BEGIN
            WriteChar['<];
            DO
                WITH e SELECT FROM
                    nil => EXIT;
                    nonNil =>
                        BEGIN
                        IF ~firstTime THEN WriteChar[',] ELSE firstTime ← FALSE;
                        PrintE[first];
                        expression ← rest;
                        END;
                    ENDCASE
                ENDLOOP;
            WriteChar['>]
            END;
        ap =>
            BEGIN WriteChar['(]; PrintE[e.rator]; WriteChar[':]; PrintE[e.rand]; WriteChar[')] END;
        ENDCASE;
    END;

PrintError: PROCEDURE[descriptor: STRING, expression: E] =
    BEGIN
    IODefs.WriteLine[descriptor]; PrintE[expression]
    END;

PrintContext: PROCEDURE[subexpression: BOOLEAN, expression: E] =
    BEGIN
    PrintError[IF subexpression THEN " in" ELSE " =", expression]
    END;

Exec: PROCEDURE =
    BEGIN
    expr: E;
    IODefs.WriteLine["Alto Mesa Red of March 20, 1979  11:20 AM"];
    DO
        BEGIN
        PrintE[Eval[expr ← ReadE[ ! SyntaxError => GO TO SyntaxError] !
                Unwind =>
                    BEGIN
                    IF subexpression THEN PrintContext[TRUE, expr];
                    IODefs.WriteLine["."];
                    CONTINUE
                    END]]
        EXITS
            SyntaxError => IODefs.WriteLine["Malformed expression"];
        END;
        IODefs.WriteLine[""]
        ENDLOOP
    END;

-- Initialization
MkAtom["and"].binding       ← primRegular[And];
MkAtom["apply"].binding     ← primRegular[RApply];
MkAtom["atom"].binding      ← primRegular[RAtom];
MkAtom["define"].binding    ← primRegular[Define];
MkAtom["distl"].binding     ← primRegular[Distl];
```

```
MkAtom["distr"].binding          ← primRegular[Distr];
MkAtom["eq"].binding             ← primRegular[Eq];
MkAtom["i"].binding              ← primRegular[I];
MkAtom["joinl"].binding          ← primRegular[Joinl];
MkAtom["not"].binding            ← primRegular[Not];
MkAtom["null"].binding           ← primRegular[Null];
MkAtom["1"].binding              ← primRegular[One];
MkAtom["or"].binding             ← primRegular[Or];
MkAtom["print"].binding          ← primRegular[Print];
MkAtom["quit"].binding           ← primRegular[Quit];
MkAtom["read"].binding           ← primRegular[Read];
MkAtom["save"].binding           ← primRegular[Save];
MkAtom["sepl"].binding           ← primRegular[Sepl];
MkAtom["tail"].binding           ← primRegular[Tail];

MkAtom["aa"].binding             ← primMeta[Aa];
MkAtom["c"].binding              ← primMeta[C];
MkAtom["cn"].binding             ← primMeta[Cn];
MkAtom["comp"].binding           ← primMeta[Comp];
MkAtom["insert"].binding         ← primMeta[Insert];
MkAtom["nf"].binding             ← primMeta[Nf];
MkAtom["while"].binding          ← primMeta[While];

Exec[]

END.
```

```
-- Red.mesa (last edited by: McJones on: 28-Oct-80 14:24:38)

DIRECTORY
  ImageDefs USING [MakeImage, StopMesa],
  IODefs USING [
    GetInputStream, ReadLine, SetInputStream, TAB, WriteChar, WriteString,
    WriteLine],
  SegmentDefs USING [FileNameError],
  StreamDefs USING [NewByteStream, Read, StreamError, StreamHandle],
  StringDefs USING [AppendChar, AppendString, EquivalentString],
  SystemDefs USING [AllocateHeapNode, AllocateHeapString];

Red: PROGRAM
  IMPORTS ImageDefs, IODefs, SegmentDefs, StreamDefs, StringDefs, SystemDefs =
  BEGIN

  EKind: TYPE = {atom, seq, ap};
  ERep: TYPE = RECORD [
    SELECT eKind: EKind FROM
      atom => [
        link: Atom,
        pname: STRING,
        binding:
          SELECT bKind: * FROM
            unbound => NULL,
            primRegular => [primitive: Regular],
            primMeta => [primitive: Meta],
            defined => [definition: E],
            ENDCASE],
      seq => [
        SELECT sKind: * FROM
          nil => NULL,
          nonNil => [first: E, rest: Seq],
          ENDCASE],
      ap => [rator: E, rand: E],
      ENDCASE];
  Atom: TYPE = POINTER TO atom ERep;
  Nil: TYPE = POINTER TO nil seq
  ERep;
  NNSeq: TYPE = POINTER TO nonNil seq
  ERep;
  Seq: TYPE = POINTER TO seq ERep;
  Ap: TYPE = POINTER TO ap ERep;
  E: TYPE = POINTER TO ERep;
  Regular: TYPE = PROC [operand: E] RETURNS [E];
  Meta: TYPE = PROC [parameters: Seq, operand: E] RETURNS [E];
  atomList: Atom ← LOOPHOLE[NIL];
  true: Atom = MkAtom["true"];
  false: Atom = MkAtom["false"];
  nilRep: nil seq ERep ← ERep[seq[nil[]]];
  nil: Nil = @nilRep;

  MkAtom: PROC [pname: STRING] RETURNS [a: Atom] =
    BEGIN
    FOR a ← atomList, a.link WHILE a ~= NIL DO
      IF StringDefs.EquivalentString[pname, a.pname] THEN EXIT
      REPEAT
        FINISHED =>
          BEGIN
```

Pair: TYPE = RECORD [first, second: E];
Triple: TYPE = RECORD [first, second, third: E];


```
          a ← LOOPHOLE[SystemDefs.AllocateHeapNode[SIZE[atom ERep]]];
          at ← ERep[
            atom[
            atomList, SystemDefs.AllocateHeapString[pname.length], unbound[]]];
          StringDefs.AppendString[a.pname, pname];
          atomList ← a;
          END;
      ENDLOOP;
    END;

  MkSeq: PROC [first: E, rest: Seq] RETURNS [s: NNSeq] = INLINE
    BEGIN
    s ← LOOPHOLE[SystemDefs.AllocateHeapNode[SIZE[nonNil seq ERep]]];
    st ← ERep[seq[nonNil[first, rest]]];
    END;

  MkAp: PROC [rator: E, rand: E] RETURNS [a: Ap] =
    BEGIN
    a ← LOOPHOLE[SystemDefs.AllocateHeapNode[SIZE[ap ERep]]];
    at ← ERep[ap[rator, rand]];
    END;

  SeqFromPair: PROC [e1, e2: E] RETURNS [NNSeq] = INLINE {
    RETURN[MkSeq[e1, MkSeq[e2, nil]]]};

  TVFromBool: PROC [b: BOOLEAN] RETURNS [Atom] = INLINE {
    RETURN[IF b THEN true ELSE false]};

  BoolFromE: PROC [expression: E] RETURNS [BOOLEAN] = INLINE
    BEGIN
    RETURN[
      SELECT AtomFromE[expression ! Error => ERROR Error["non truth value"]] FROM
        true => TRUE,
        false => FALSE,
        ENDCASE => ERROR Error["non truth value"]]
    END;

  UnitFromSeq: PROC [sequence: Seq] RETURNS [E] = INLINE
    BEGIN
    nns: NNSeq = NNSeqFromSeq[sequence];
    RETURN[IF nns.rest = nil THEN nns.first ELSE ERROR Error["too long"]]
    END;

  PairFromSeq: PROC [sequence: Seq] RETURNS [E, E] = INLINE
    BEGIN
    nns: NNSeq = NNSeqFromSeq[sequence];
    RETURN[nns.first, UnitFromSeq[nns.rest]]
    END;

  TripleFromSeq: PROC [sequence: Seq] RETURNS [E, E, E] = INLINE
    BEGIN
    nns: NNSeq = NNSeqFromSeq[sequence];
    e2, e3: E;
    [e2, e3] ← PairFromSeq[nns.rest];
    RETURN[nns.first, e2, e3]
    END;

  AtomFromE: PROC [expression: E] RETURNS [Atom] = INLINE
    BEGIN
```

Formatting AB: leading

pair: Pair
pair.first
pair.second
Pair
triple:
Triple
triple.first or nns.first
triple.third

```
  RETURN[
    WITH e: expression SELECT FROM
      atom => @e,
      ENDCASE => ERROR Error["non atom"]]
  END;

SeqFromE: PROC [expression: E] RETURNS [Seq] = INLINE
  BEGIN
  RETURN[
    WITH e: expression SELECT FROM seq => @e, ENDCASE => ERROR Error["non seq"]]
  END;

NNSeqFromSeq: PROC [sequence: Seq] RETURNS [NNSeq] = INLINE
  BEGIN
  RETURN[
    WITH s: sequence SELECT FROM
      nil => ERROR Error["too short"],
      nonNil => @s,
      ENDCASE => ERROR]
  END;

NNSeqFromE: PROC [expression: E] RETURNS [NNSeq] = INLINE {
  RETURN[NNSeqFromSeq[SeqFromE[expression]]]};

PairFromE: PROC [expression: E] RETURNS [e1, e2: E] = INLINE {
  [e1, e2] ← PairFromSeq[SeqFromE[expression]]; RETURN[e1, e2]};

MapSeq: PROC [sequence: Seq, f: PROC [E] RETURNS [E]] RETURNS [result: Seq] =
  BEGIN
  old, new: NNSeq;
  WITH s: sequence SELECT FROM
    nil => result ← sequence;
    nonNil =>
      BEGIN
      old ← @s;
      new ← result ← MkSeq[f[old.first], nil];
      DO
        WITH r: old.rest SELECT FROM nil => EXIT; nonNil => old ← @r; ENDCASE;
        new ← new.rest ← MkSeq[f[old.first], nil];
        ENDLOOP;
      END;
    ENDCASE;
  END;

Eval: PROC [expression: E] RETURNS [E] =
  BEGIN
  RETURN[
    WITH e: expression SELECT FROM
      atom => expression,
      seq => MapSeq[@e, Eval ! Unwind => ERROR Unwind[TRUE]],
      ap => Apply[Eval[e.rator], Eval[e.rand]],
      ENDCASE => ERROR]
  END;

Apply: PROC [operator: E, operand: E] RETURNS [E] =
  BEGIN
  metaRator, metaRand: E;
  BEGIN
  ENABLE
    BEGIN
```



```
    Error =>
      BEGIN
      PrintError[descriptor, MkAp[operator, operand]];
      GO TO StartUnwind
      END;
    Unwind =>
      BEGIN
      PrintContext[subexpression, MkAp[operator, operand]];
      ERROR Unwind[FALSE]
      END
    END;
  WITH rator: operator SELECT FROM
    atom =>
      WITH atomRator: rator SELECT FROM
        unbound => ERROR Error["unbound atom"];
        primRegular => RETURN[atomRator.primitive[operand]];
        primMeta =>
          BEGIN
          [metaRator, metaRand] ← PairFromE[operand];
          RETURN[atomRator.primitive[NNSeqFromE[metaRator], metaRand]]
          END;
        defined => RETURN[Apply[atomRator.definition, operand]];
        ENDCASE;
    seq =>
      WITH seqRator: rator SELECT FROM
        nil => ERROR Error["nil seq operator"];
        nonNil =>
          WITH srf: seqRator.first SELECT FROM
            atom =>
              WITH srf SELECT FROM
                unbound => ERROR Error["unbound atom"];
                primRegular =>
                  RETURN[primitive[SeqFromPair[@seqRator, operand]]];
                primMeta => RETURN[primitive[seqRator.rest, operand]];
                defined =>
                  RETURN[Apply[MkSeq[definition, seqRator.rest], operand]];
                ENDCASE;
            seq => RETURN[Apply[@srf, SeqFromPair[@seqRator, operand]]];
            ENDCASE => ERROR;
        ENDCASE;
    ENDCASE => ERROR;
  EXITS StartUnwind => ERROR Unwind[FALSE];
  END;
  ERROR
  END;

Error: ERROR [descriptor: STRING] = CODE;
Unwind: ERROR [subexpression: BOOLEAN] = CODE;

And: Regular =
  BEGIN
  e1, e2: E;
  [e1, e2] ← PairFromE[operand];
  RETURN[TVFromBool[BoolFromE[e1] AND BoolFromE[e2]]]
  END;

RApply: Regular =
  BEGIN
  rator, rand: E;
```

```
  [rator, rand] ← PairFromE[operand];
  RETURN[Apply[rator, rand]]
  END;

RAtom: Regular = {RETURN[TVFromBool[operand.eKind = atom]]};

Define: Regular =
  BEGIN
  name, value: E;
  n: Atom;
  [name, value] ← PairFromE[operand];
  n ← AtomFromE[name ! Error => ERROR Error["non atom first"]];
  WITH v: value SELECT FROM
    atom =>
      WITH v SELECT FROM -- in lieu of entire variant as RightSide:
        unbound => n.binding ← unbound[];
        primRegular => n.binding ← primRegular[primitive];
        primMeta => n.binding ← primMeta[primitive];
        defined => n.binding ← defined[definition];
        ENDCASE;
    seq => n.binding ← defined[@v];
    ENDCASE => ERROR;
  RETURN[operand]
  END;

DistI: Regular =
  BEGIN
  first, second: E;
  PairL: PROC [e: E] RETURNS [E] = {RETURN[SeqFromPair[first, e]]};
  [first, second] ← PairFromE[operand];
  RETURN[MapSeq[SeqFromE[second], PairL]]
  END;

Distr: Regular =
  BEGIN
  first, second: E;
  PairR: PROC [e: E] RETURNS [E] = {RETURN[SeqFromPair[e, second]]};
  [first, second] ← PairFromE[operand];
  RETURN[MapSeq[SeqFromE[first], PairR]]
  END;

Eq: Regular =
  BEGIN
  EqPair: PROC [expression1, expression2: E] RETURNS [BOOLEAN] =
    BEGIN
    seq1, seq2: Seq;
    WITH e1: expression1 SELECT FROM
      atom =>
        RETURN[
          WITH e2: expression2 SELECT FROM
            atom => @e1 = @e2,
            seq => FALSE,
            ENDCASE => ERROR];
      seq =>
        WITH e2: expression2 SELECT FROM
          atom => RETURN[FALSE];
          seq =>
            BEGIN
```



```
            seq1 ← @e1;
            seq2 ← @e2;
            DO
              WITH s1: seq1 SELECT FROM
                nil =>
                  RETURN[
                    WITH s2: seq2 SELECT FROM nil => TRUE, ENDCASE => FALSE];
                nonNil =>
                  WITH s2: seq2 SELECT FROM
                    nil => RETURN[FALSE];
                    nonNil =>
                      BEGIN
                      IF ~EqPair[s1.first, s2.first] THEN RETURN[FALSE];
                      seq1 ← s1.rest;
                      seq2 ← s2.rest
                      END;
                    ENDCASE;
                ENDCASE;
              ENDLOOP;
            END;
          ENDCASE => ERROR;
      ENDCASE => ERROR
    END;
  e1, e2: E;
  [e1, e2] ← PairFromE[operand];
  RETURN[TVFromBool[EqPair[e1, e2]]]
  END;

I: Regular = {RETURN[operand]};

JoinI: Regular =
  BEGIN
  first, second: E;
  [first, second] ← PairFromE[operand];
  RETURN[
    MkSeq[first, SeqFromE[second ! Error => ERROR Error["non seq second"]]]]
  END;

Not: Regular = {RETURN[TVFromBool[NOT BoolFromE[operand]]]};

Null: Regular = {RETURN[TVFromBool[operand = nil]]};

One: Regular = {RETURN[NNSeqFromE[operand].first]};

Print: Regular = {PrintE[operand]; IODefs.WriteLine[""]; RETURN[operand]};

Quit: Regular = {ImageDefs.StopMesa[]; ERROR};

Read: Regular =
  BEGIN
  a: Atom = AtomFromE[operand];
  strmOld: StreamDefs.StreamHandle = IODefs.GetInputStream[];
  strmNew: StreamDefs.StreamHandle = StreamDefs.NewByteStream[
    name: a.pname, access: StreamDefs.Read !
    SegmentDefs.FileNameError => ERROR Error["nonexistent file to read"]];
  CleanUp: PROC = {IODefs.SetInputStream[strmOld]; strmNew.destroy[strmNew]};
  e: E;
  BEGIN
  ENABLE UNWIND => CleanUp[];
```

Red.mesa. 28-Oct-80 15:45:50 5

Red.mesa. 28-Oct-80 15:45:50 6

```
  IODefs.SetInputStream[strmNew];
  e ← ReadE[
    !
    SyntaxError, StreamDefs.StreamError =>
      ERROR Error["syntax error in file to read"]];
  CleanUp[]
  END;
  RETURN[Eval[e]]
  END;

Save: Regular = {
  ImageDefs.MakeImage[AtomFromE[operand].pname]; RETURN[operand]};

Sepl: Regular =
  BEGIN
  nns: NNSeq = NNSeqFromE[operand];
  RETURN[SeqFromPair[nns.first, nns.rest]]
  END;

Tail: Regular = {RETURN[NNSeqFromE[operand].rest]};

Aa: Meta =
  BEGIN
  f: E = UnitFromSeq[parameters ! Error => Error["wrong number of parameters"]];
  ApplyToElt: PROC [e: E] RETURNS [E] = {
    RETURN[Apply[f, e ! Unwind => ERROR Unwind[TRUE]]]};
  RETURN[MapSeq[SeqFromE[operand], ApplyToElt]]
  END;

C: Meta = {
  RETURN[
    UnitFromSeq[parameters ! Error => Error["wrong number of parameters"]]]};

Cn: Meta =
  BEGIN
  p, f, g: E;
  [p, f, g] ← TripleFromSeq[
    parameters ! Error => Error["wrong number of parameters"]];
  RETURN[
    Apply[
      IF BoolFromE[Apply[p, operand ! Unwind => ERROR Unwind[TRUE]]] THEN f
      ELSE g, operand]]
  END;

Comp: Meta =
  BEGIN
  RETURN[
    WITH parameters SELECT FROM
      nil => operand,
      nonNil => Apply[
        first, Comp[rest, operand] ! Unwind => ERROR Unwind[TRUE]],
      ENDCASE => ERROR]
  END;

Insert: Meta =
  BEGIN
  f: E = UnitFromSeq[parameters];
  nns: NNSeq ← NNSeqFromE[operand];
  result: E ← nns.first;
```



```
  DO
    ENABLE Unwind => ERROR Unwind[TRUE];
    WITH tail: nns.rest SELECT FROM
      nil => RETURN[result];
      nonNil =>
        BEGIN
        result ← Apply[f, SeqFromPair[result, tail.first]];
        nns ← @tail;
        END;
      ENDCASE;
    ENDLOOP;
  END;

Nf: Meta =
  BEGIN
  ApplyEltTo: PROC [f: E] RETURNS [E] = {
    RETURN[Apply[f, operand ! Unwind => ERROR Unwind[TRUE]]]};
  RETURN[MapSeq[parameters, ApplyEltTo]]
  END;

While: Meta =
  BEGIN
  result: E ← operand;
  p, f: E;
  [p, f] ← PairFromSeq[
    parameters ! Error => ERROR Error["wrong number of parameters"]];
  BEGIN
  ENABLE Unwind => ERROR Unwind[TRUE];
  WHILE BoolFromE[Apply[p, result]] DO result ← Apply[f, result] ENDLOOP;
  RETURN[result]
  END
  END;

TKind: TYPE = {atom, beginSeq, delimSeq, endSeq, beginAp, delimAp, endAp};
Token: TYPE = RECORD [
  SELECT kind: TKind FROM
    atom => [body: Atom],
    beginSeq, delimSeq, endSeq, beginAp, delimAp, endAp => NULL,
    ENDCASE];

ReadTokenLine: STRING = [200];
ReadTokenLineEmpty: BOOLEAN ← TRUE;
ReadTokenLineIndex: CARDINAL;
ReadTokenString: STRING = [40];
ReadTokenToken: Token;
ReadTokenTokenPresent: BOOLEAN ← FALSE;

ReadE: PROC RETURNS [expression: E] =
  BEGIN
  s: NNSeq;
  f, a: E;
  ReadToken: PROC RETURNS [t: Token] =
    BEGIN
    ch: CHARACTER;
    ReadChar: PROC RETURNS [ch: CHARACTER] =
      BEGIN
      WHILE ReadTokenLineEmpty OR ReadTokenLineIndex = ReadTokenLine.length DO
        IODefs.ReadLine[ReadTokenLine];
        ReadTokenLineIndex ← 0;
        ReadTokenLineEmpty ← FALSE
```

```
    ENDLOOP;
  ch ← ReadTokenLine[ReadTokenLineIndex];
  ReadTokenLineIndex ← ReadTokenLineIndex + 1;
  RETURN[ch]
  END;
UnreadChar: PROC = {ReadTokenLineIndex ← ReadTokenLineIndex - 1};
-- Body of ReadToken
IF ReadTokenTokenPresent THEN
  BEGIN ReadTokenTokenPresent ← FALSE; RETURN[ReadTokenToken] END;
DO
  SELECT ch ← ReadChar[] FROM
    ' , IODefs.TAB => NULL;
    '< => BEGIN t ← Token[beginSeq[]]; EXIT END;
    ', => BEGIN t ← Token[delimSeq[]]; EXIT END;
    '> => BEGIN t ← Token[endSeq[]]; EXIT END;
    '( => BEGIN t ← Token[beginAp[]]; EXIT END;
    ': => BEGIN t ← Token[delimAp[]]; EXIT END;
    ') => BEGIN t ← Token[endAp[]]; EXIT END;
    ENDCASE =>
      BEGIN
      ReadTokenString.length ← 1;
      ReadTokenString[0] ← ch;
      DO
        SELECT ch ← ReadChar[] FROM
          ' , IODefs.TAB, '<, ',, '>, '(, ':, ') => EXIT;
          ENDCASE => StringDefs.AppendChar[ReadTokenString, ch]
        ENDLOOP;
      UnreadChar;
      t ← Token[atom[MkAtom[ReadTokenString]]];
      EXIT
      END
  ENDLOOP;
ReadTokenToken ← t;
RETURN[t]
END;
UnreadToken: PROC = {
  IF ReadTokenTokenPresent THEN ERROR; ReadTokenTokenPresent ← TRUE};
t: Token ← ReadToken[]; -- Body of ReadE
WITH t SELECT FROM
  atom => expression ← body;
  beginSeq =>
    BEGIN
    IF ReadToken[].kind = endSeq THEN RETURN[nil] ELSE UnreadToken[];
    expression ← s ← MkSeq[ReadE[], nil];
    DO
      SELECT ReadToken[].kind FROM
        delimSeq => s ← s.rest ← MkSeq[ReadE[], nil];
        endSeq => EXIT;
        ENDCASE => ERROR SyntaxError
      ENDLOOP
    END;
  beginAp =>
    BEGIN
    f ← ReadE[];
    IF ReadToken[].kind ~= delimAp THEN ERROR SyntaxError;
    a ← ReadE[];
    IF ReadToken[].kind ~= endAp THEN ERROR SyntaxError;
    expression ← MkAp[f, a]
    END;
  ENDCASE => ERROR SyntaxError;
```



```
  RETURN[expression]
  END;

SyntaxError: ERROR = CODE;

PrintE: PROC [expression: E] =
  BEGIN OPEN IODefs;
  firstTime: BOOLEAN ← TRUE;
  WITH e: expression SELECT FROM
    atom => WriteString[e.pname];
    seq =>
      BEGIN
      WriteChar['<];
      DO
        WITH e SELECT FROM
          nil => EXIT;
          nonNil =>
            BEGIN
            IF ~firstTime THEN WriteChar[',] ELSE firstTime ← FALSE;
            PrintE[first];
            expression ← rest;
            END;
        ENDCASE
      ENDLOOP;
      WriteChar['>]
      END;
    ap =>
      BEGIN
      WriteChar['(];
      PrintE[e.rator];
      WriteChar[':];
      PrintE[e.rand];
      WriteChar[')]
      END;
    ENDCASE;
  END;

PrintError: PROC [descriptor: STRING, expression: E] = {
  IODefs.WriteLine[descriptor]; PrintE[expression]};

PrintContext: PROC [subexpression: BOOLEAN, expression: E] = {
  PrintError[IF subexpression THEN " in" ELSE " = ", expression]};

Exec: PROC =
  BEGIN
  expr: E;
  IODefs.WriteLine["Alto Mesa Red of 28-Oct-80 14:24:38"];
  DO
    BEGIN
    PrintE[
      Eval[
      expr ← ReadE[ ! SyntaxError => GO TO SyntaxError] !
      Unwind =>
        BEGIN
        IF subexpression THEN PrintContext[TRUE, expr];
        IODefs.WriteLine["."];
        CONTINUE
        END]]
    EXITS SyntaxError => IODefs.WriteLine["Malformed expression"];
    END;
```

```
    IODefs.WriteLine[""]
    ENDLOOP
  END;

--
-- Initialization

MkAtom["and"].binding ← primRegular[And];
MkAtom["apply"].binding ← primRegular[RApply];
MkAtom["atom"].binding ← primRegular[RAtom];
MkAtom["define"].binding ← primRegular[Define];
MkAtom["distl"].binding ← primRegular[Distl];
MkAtom["distr"].binding ← primRegular[Distr];
MkAtom["eq"].binding ← primRegular[Eq];
MkAtom["i"].binding ← primRegular[I];
MkAtom["joinl"].binding ← primRegular[Joinl];
MkAtom["not"].binding ← primRegular[Not];
MkAtom["null"].binding ← primRegular[Null];
MkAtom["1"].binding ← primRegular[One];
MkAtom["print"].binding ← primRegular[Print];
MkAtom["quit"].binding ← primRegular[Quit];
MkAtom["read"].binding ← primRegular[Read];
MkAtom["save"].binding ← primRegular[Save];
MkAtom["sepl"].binding ← primRegular[Sepl];
MkAtom["tail"].binding ← primRegular[Tail];
MkAtom["aa"].binding ← primMeta[Aa];
MkAtom["c"].binding ← primMeta[C];
MkAtom["cn"].binding ← primMeta[Cn];
MkAtom["comp"].binding ← primMeta[Comp];
MkAtom["insert"].binding ← primMeta[Insert];
MkAtom["nf"].binding ← primMeta[Nf];
MkAtom["while"].binding ← primMeta[While];

Exec[];

END.
```

```
<
<or, <comp, not, and, <aa, not>>>
,
<2, <comp, 1, tail>>
,
<3, <comp, 1, tail, tail>>
,
<lasti, <comp, 1, <while, <comp, not, null, tail>, tail>>>
,
<lastr, <cn, <comp, null, tail>, 1, <comp, lastr, tail>>>
,
<lastm, <<cn, <comp, null, tail, 2>,
              <comp, 1, 2>,
              <comp, apply, <nf, 1, <comp, tail, 2>>>>>>
,
<joinr, <<cn, <comp, null, 1, 2>,
              <nf, <comp, 2, 2>>,
              <comp, joinl, <nf, <comp, 1, 1, 2>,
                 <comp, apply,
                    <nf, 1, <nf, <comp, tail, 1, 2>,
                            <comp, 2, 2>>>>>>>>>
,
<reverse, <comp, 2,
                 <while, <comp, not, null, 1>,
                         <nf, <comp, tail, 1>,
                              <comp, joinl, <nf, <comp, 1, 1>, 2>>>>,
                 <nf, i, <nf>>>>>
,
<cat, <comp, 1, <while, <comp, not, null, 2>,
                        <nf, <comp, joinr, <nf, 1, <comp, 1, 2>>>,
                             <comp, tail, 2>>>>>
,
<trans, <comp, 1,
              <while, <comp, not, null, 1, 2>,
                      <nf, <comp, joinr, <nf, 1, <comp, <aa, 1>, 2>>>,
                                         <comp, <aa, tail>, 2>>>,
              <nf, <nf>, i>>>
,
<zerop, null>
,
<plus1, <comp, joinl, <nf, <c, 1>, i>>>
,
<sub1, tail>
,
<sum, <comp, 2,
             <while, <comp, not, zerop, 1>,
                     <nf, <comp, sub1, 1>, <comp, plus1, 2>>>>>
,
<product,
 <comp,
  3,
  <while,
   <comp, not, zerop, 1>,
   <nf, <comp, sub1, 1>, 2, <comp, sum, <nf, 2, 3>>>>,
  <nf, 1, 2, <c, <>>>>>
,
<square, <comp, product, <nf, i, i>>>
,
<exp,
 <comp,
  3,
  <while,
   <comp, not, zerop, 1>,
   <nf, <comp, sub1, 1>, 2, <comp, product, <nf, 2, 3>>>>,
  <nf, 1, 2, <c, <1>>>>>
,
<n2n, <comp, exp, <nf, i, i>>>
,
<fact, <<cn, <comp, zerop, 2>,
             <c, <1>>,
             <comp, product,
                    <nf, 2,
                         <comp, apply, <nf, 1, <comp, sub1, 2>>>>>>>>>
>
```

```
Alto/Tajo 6.0 of 27-Oct-80 17:06
28-Oct-80 15:30
>Red -- 141424B
Alto Mesa Red of 28-Oct-80 14:24:38

(<comp, <aa,define>, read>:defs.red)
<
<or, <comp, not, and, <aa, not>>>
,
<2, <comp, 1, tail>>
,
<3, <comp, 1, tail, tail>>
,
<lasti, <comp, 1, <while, <comp, not, null, tail>, tail>>>
,
<lastr, <cn, <comp, null, tail>, 1, <comp, lastr, tail>>>
,
<lastm, <<cn, <comp, null, tail, 2>,
              <comp, 1, 2>,
              <comp, apply, <nf, 1, <comp, tail, 2>>>>>>
,
<joinr, <<cn, <comp, null, 1, 2>,
              <nf, <comp, 2, 2>>,
              <comp, joinl, <nf, <comp, 1, 1, 2>,
                 <comp, apply,
                    <nf, 1, <nf, <comp, tail, 1, 2>,
                          <comp, 2, 2>>>>>>>>>
,
<reverse, <comp, 2,
              <while, <comp, not, null, 1>,
                      <nf, <comp, tail, 1>,
                           <comp, joinl, <nf, <comp, 1, 1>, 2>>>>,
              <nf, i, <nf>>>>
,
<cat, <comp, 1, <while, <comp, not, null, 2>,
                        <nf, <comp, joinr, <nf, 1, <comp, 1, 2>>>,
                             <comp, tail, 2>>>>>
,
<trans, <comp, 1,
              <while, <comp, not, null, 1, 2>,
                      <nf, <comp, joinr, <nf, 1, <comp, <aa, 1>, 2>>>,
                                         <comp, <aa, tail>, 2>>>,
              <nf, <nf>, i>>>
,
<zerop, null>
,
<plus1, <comp, joinl, <nf, <c, 1>, i>>>
,
<sub1, tail>
,
<sum, <comp, 2,
              <while, <comp, not, zerop, 1>,
                      <nf, <comp, sub1, 1>, <comp, plus1, 2>>>>>
,
<product,
 <comp,
  3,
  <while,
   <comp, not, zerop, 1>,
   <nf, <comp, sub1, 1>, 2, <comp, sum, <nf, 2, 3>>>>,
  <nf, 1, 2, <c, <>>>>>
,
<square, <comp, product, <nf, i, i>>>
,
<exp,
 <comp,
  3,
  <while,
   <comp, not, zerop, 1>,
   <nf, <comp, sub1, 1>, 2, <comp, product, <nf, 2, 3>>>>,
  <nf, 1, 2, <c, <1>>>>>
,
<n2n, <comp, exp, <nf, i, i>>>
,
<fact, <<cn, <comp, zerop, 2>,
             <c, <1>>,
```

```
            <comp, product,
                  <nf, 2,
                        <comp, apply, <nf, 1, <comp, sub1, 2>>>>>>>>
>
<<or,<comp,not,and,<aa,not>>>,<2,<comp,1,tail>>,<3,<comp,1,tail,tail>>,<lasti,<comp,1,<while,<comp,not,
**null,tail>,tail>>>,<lastr,<cn,<comp,null,tail>,1,<comp,lastr,tail>>>,<lastm,<<cn,<comp,null,tail,2>,<
**comp,1,2>,<comp,apply,<nf,1,<comp,tail,2>>>>>>,<joinr,<<cn,<comp,null,1,2>,<nf,<comp,2,2>>,<comp,join
**l,<nf,<comp,1,1,2>,<comp,apply,<nf,1,<nf,<comp,tail,1,2>,<comp,2,2>>>>>>>>>,<reverse,<comp,2,<while,<
**comp,not,null,1>,<nf,<comp,tail,1>,<comp,joinl,<nf,<comp,1,1>,2>>>>,<nf,i,<nf>>>>,<cat,<comp,1,<while
**,<comp,not,null,2>,<nf,<comp,joinr,<nf,1,<comp,1,2>>>,<comp,tail,2>>>>>,<trans,<comp,1,<while,<comp,n
**ot,null,1,2>,<nf,<comp,joinr,<nf,1,<comp,<aa,1>,2>>>,<comp,<aa,tail>,2>>>,<nf,<nf>,i>>>,<zerop,null>,
**<plus1,<comp,joinl,<nf,<c,1>,i>>>,<sub1,tail>,<sum,<comp,2,<while,<comp,not,zerop,1>,<nf,<comp,sub1,1
**>,<comp,plus1,2>>>>>,<product,<comp,3,<while,<comp,not,zerop,1>,<nf,<comp,sub1,1>,2,<comp,sum,<nf,2,3
**>>>>,<nf,1,2,<c,<>>>>>,<square,<comp,product,<nf,i,i>>>,<exp,<comp,3,<while,<comp,not,zerop,1>,<nf,<c
**omp,sub1,1>,2,<comp,product,<nf,2,3>>>>,<nf,1,2,<c,<1>>>>>,<n2n,<comp,exp,<nf,i,i>>>,<fact,<<cn,<comp
**,zerop,2>,<c,<1>>,<comp,product,<nf,2,<comp,apply,<nf,1,<comp,sub1,2>>>>>>>>>>
(or:<true,false>)
true
(2: <a,b,c>)
b
(3: <a,b,c>)
c
(lasti:<a,b,c,d,e>)
e
(Lasti:<a,b,c,d,e,f>)
f
(lastr:<a,b,c,d,e,f>)
f
<joinr:<<a,b,c>,d>)
Malformed expression

<<a,b,c>,d>
Malformed expression

(joinr:<<a,b,c,>,d>)
Malformed expression

Malformed expression

d
Malformed expression

Malformed expression

(joinr:<<a,b,c>,d>)
unbound atom
(<a,b,c>:<<<b,c>,d>,<<b,c>,d>>) =
(<<a,b,c>,d>:<<b,c>,d>) =
(apply:<<<a,b,c>,d>,<<b,c>,d>>) in
(<comp,apply,<nf,1,<nf,<comp,tail,1,2>,<comp,2,2>>>>:<<<a,b,c>,d>,<<a,b,c>,d>>) in
(<nf,<comp,1,1,2>,<comp,apply,<nf,1,<nf,<comp,tail,1,2>,<comp,2,2>>>>>:<<<a,b,c>,d>,<<a,b,c>,d>>) in
(<comp,joinl,<nf,<comp,1,1,2>,<comp,apply,<nf,1,<nf,<comp,tail,1,2>,<comp,2,2>>>>>>:<<<a,b,c>,d>,<<a,b,
**c>,d>>) =
(<cn,<comp,null,1,2>,<nf,<comp,2,2>>,<comp,joinl,<nf,<comp,1,1,2>,<comp,apply,<nf,1,<nf,<comp,tail,1,2>
**,<comp,2,2>>>>>>>:<<<a,b,c>,d>,<<a,b,c>,d>>) =
defined (<<cn,<comp,null,1,2>,<nf,<comp,2,2>>,<comp,joinl,<nf,<comp,1,1,2>,<comp,apply,<nf,1,<nf,<comp,tail,1,2
**>,<comp,2,2>>>>>>>>:<<a,b,c>,d>) =
(joinr:<<a,b,c>,d>).

(joinl:<<a,b>,<c,d>>)
<<a,b>,c,d>
(joinr:<<a,b>,<c,d>>)
unbound atom
(<a,b>:<<<b>,<c,d>>,<<b>,<c,d>>>) =
(<<a,b>,<c,d>>:<<b>,<c,d>>) =
(apply:<<<a,b>,<c,d>>,<<b>,<c,d>>>) in
(<comp,apply,<nf,1,<nf,<comp,tail,1,2>,<comp,2,2>>>>:<<<a,b>,<c,d>>,<<a,b>,<c,d>>>) in
(<nf,<comp,1,1,2>,<comp,apply,<nf,1,<nf,<comp,tail,1,2>,<comp,2,2>>>>>:<<<a,b>,<c,d>>,<<a,b>,<c,d>>>) i
**n
(<comp,joinl,<nf,<comp,1,1,2>,<comp,apply,<nf,1,<nf,<comp,tail,1,2>,<comp,2,2>>>>>>:<<<a,b>,<c,d>>,<<a,
**b>,<c,d>>>) =
(<cn,<comp,null,1,2>,<nf,<comp,2,2>>,<comp,joinl,<nf,<comp,1,1,2>,<comp,apply,<nf,1,<nf,<comp,tail,1,2>
**,<comp,2,2>>>>>>>:<<<a,b>,<c,d>>,<<a,b>,<c,d>>>) =
(<<cn,<comp,null,1,2>,<nf,<comp,2,2>>,<comp,joinl,<nf,<comp,1,1,2>,<comp,apply,<nf,1,<nf,<comp,tail,1,2
**>,<comp,2,2>>>>>>>>:<<a,b>,<c,d>>) =
(joinr:<<a,b>,<c,d>>).
```

```
(reverse:<a,b,c,d>)
<d,c,b,a>
(cat:<<a,b>,<c,d>>)
unbound atom
(<a,b>:<<<b>,c>,<<b>,c>>) =
(<<a,b>,c>:<<b>,c>) =
(apply:<<<a,b>,c>,<<b>,c>>) in
(<comp,apply,<nf,1,<nf,<comp,tail,1,2>,<comp,2,2>>>>:<<<a,b>,c>,<<a,b>,c>>) in
(<nf,<comp,1,1,2>,<comp,apply,<nf,1,<nf,<comp,tail,1,2>,<comp,2,2>>>>>:<<<a,b>,c>,<<a,b>,c>>) in
(<comp,joinl,<nf,<comp,1,1,2>,<comp,apply,<nf,1,<nf,<comp,tail,1,2>,<comp,2,2>>>>>>:<<<a,b>,c>,<<a,b>,c
**>>) =
(<cn,<comp,null,1,2>,<nf,<comp,2,2>>,<comp,joinl,<nf,<comp,1,1,2>,<comp,apply,<nf,1,<nf,<comp,tail,1,2>
**,<comp,2,2>>>>>>:<<<a,b>,c>,<<a,b>,c>>) =
(<<cn,<comp,null,1,2>,<nf,<comp,2,2>>,<comp,joinl,<nf,<comp,1,1,2>,<comp,apply,<nf,1,<nf,<comp,tail,1,2
**>,<comp,2,2>>>>>>>:<<a,b>,c>) =
(joinr:<<a,b>,c>) in
(<comp,joinr,<nf,1,<comp,1,2>>>:<<a,b>,<c,d>>) in
(<nf,<comp,joinr,<nf,1,<comp,1,2>>>,<comp,tail,2>>:<<a,b>,<c,d>>) in
(<while,<comp,not,null,2>,<nf,<comp,joinr,<nf,1,<comp,1,2>>>,<comp,tail,2>>>:<<a,b>,<c,d>>) in
(<comp,1,<while,<comp,not,null,2>,<nf,<comp,joinr,<nf,1,<comp,1,2>>>,<comp,tail,2>>>>:<<a,b>,<c,d>>) =
(cat:<<a,b>,<c,d>>).

(exp:<<1,1>,<1,1,1>>)
<1,1,1,1,1,1,1,1,1>
(exp:<<1,1,1>,<1,1>>)
<1,1,1,1,1,1,1,1>
(quit:x)
```